The Thunder C Compiler

Thunder Software

The THUNDER C Compiler

USERS GUIDE

by

John T. Cox

Copyright 1984 by John T. Cox and Thunder Software
Copyright 1979,1980 by APPLE Computer Inc. and Copyright 1979
by the Regents of California.

THUNDER C COMPILER
USERS GUIDE

CONTENTS

COPYRIGHT PAGE and TITLE PAGE 1
DEDICATION 3
INTRODUCTION 4
SCOPE OF THE DOCUMENTATION 5
WHAT IS C 6
DIFFERENCES AND CAUTIONS 8
USING THE COMPILER 10
THE STDIO LIBRARY 13
THE UTILITY LIBRARY 14
ASSEMBLING YOUR COMPILED PROGRAM 15
LINKING TO THE DUMMY PROGRAM 16
LINKING TO YOUR OWN PROGRAMS 17
EXECUTION OF YOUR PROGRAM 18

APPENDICES
A. C SYNTAX SUPPORTED ON THIS COMPILER 20
B. THE PREPROCESSOR 21
C. SOME EXAMPLES 22
D. OVERVIEW OF THE COMPILER 23
E. MEMORY MAP DURING FUNCTION CALLS 24

Special appreciation to KERRY-DAWN for being the finest person I have ever met and the most loved person in my life, my wife.

The C programming language is by far the most powerful language available today because of its machine independence and its 'universal' ability to manipulate a machine directly without being specific to that particular environment. C has been described as a portable assembler which is very close to being a reality.

The implementation of C that you are about to learn is a major subset of C and therefore lacks a few of the features found in full C. This should not cause difficulty with anyone but the number crunchers among us who may miss floating point math. The original creation of this compiler was by RON CAIN and was published in Dr. Dobbs Journal several years ago. A new version with enhanced capabilities was published in early 1983 by JIM HENDRIX in Dr. Dobbs Journal. The THUNDER compiler is a close cousin to the Hendrix version. It is rewritten for APPLE Pascal and several modifications and improvements have been made. The ability to bring this compiler to you at the low price is due to the hard work of these two gentlemen and their donation of the source code to the public domain. I applaud them.

This is your compiler. What this means is that any programs you write or comments you make may be applied to the product. You can have a say in where this compiler is going for its next release. But only if you contact us and tell us what you are thinking. Remember that no one person can think up all of the good ideas or see all of the problems in a product. It takes teamwork and you are part of the team. Enjoy your new compiler and we will be waiting to hear from you.

John T. Cox, February 1984.
Thunder Software

THE SCOPE OF THIS DOCUMENT

This booklet is an explanation of the operation of the C compiler implemented here and is not a tutorial in C. A brief description of the C language will be included for reference purposes.

The APPLE Pascal system will not be explained other than where specific points need to be brought out. The required programs for operation of the compiler are explained later. It is recommended that you have the reference books The APPLE Pascal Reference Manual and The APPLE Pascal Operating System Reference Manual handy for the descriptions of the SYSTEM.LINKER and the SYSTEM.-ASSEMBLER programs.

What we do intend to convey in this booklet is how to use the C compiler in the UCSD/APPLE Pascal environment and the differences between our implementation of C and the standard defined by Kernighan and Ritchie. Along the way we hope to explain compilers in general and this compiler specifically.

The C language is becoming a very popular language to program in. Many microcomputers are getting C and UNIX capabilities which brings a great deal of power to the level of the micro. It is well known among C afficionados that most of the operating system UNIX is written in C. Byte magazine devoted a whole issue to the C language. Several leading software products have been written in C as well. So what is this language all about?

The language was developed by Mr. Dennis Ritchie (see the book THE C PROGRAMMING LANGUAGE by Kernighan and Ritchie) and was designed to be the language of the UNIX operating system. Kernighan and Ritchie state that of 13000 lines of code in the UNIX operating system itself only 800 are written in the assembly language of the host computer! Even so, C should not be looked at as only a systems programming language because you may use it to write nearly anything from word processing to FOURIER transforms.

The features that C has that people like are: its structured statement forms (I define this as the ability to program complex algorithms without goto statements. The usefulness of the goto is far below the mucking up of program flow. The ease of structuring a programs logic is exemplified by the fact that the entire compiler has no goto tatements in over 3500 lines of code originally written in C and now in Pascal.), the notion of local and global variables in functions (unlike BASIC which is entirely global and FORTRAN which really doesn't have global variables), the ability to deal with machine memory directly without the need of programming in a specific machine language, its compactness and its universality or standardization and lastly, its unique and powerful operator set allowing more control over the host machine than just about any other language. It is certain that C adherents could add line after line to the brief list above but these few statements really sum up the best of C.

The capabilities of any programming language can only be appreciated by one who uses it and pushes towards the limit of the languages design. It is recommended that the reader purchase one of the tutorial books on C as well as the book THE C PROGRAMMING LANGUAGE by Kernighan and Ritchie, Prentice Hall, Englewood Cliffs New Jersey 07632 .

For those just now embarking into the realm of C here is an example program that prints the alphabet:

```
alpha() { char p;
   p = 'A';
   while (p<='Z') putc(p++);
   }
```

Alpha is the functions name. The () are required even though there are no arguments. Each block of code begins with { and ends with }. Our character variable p is local to the function and is initialized to be the letter A. The syntax of the WHILE statement is

```
    WHILE (expression) {block}
or  WHILE (expression)  expression;
```

So, while p is less than or equal to Z, the function alpha will call another function putc. Putc is a standard library function to put a character on the screen. Finally something strange! What is the meaning of p++? Two plusses don't normally mean anything to humans. C allows this as a shorthand for the expression p=p+1; The fact that we have written the double plus AFTER the variable instead of BEFORE it (like this ++p) shows that we want 1 to be added to p after we have called the function putc which uses it.

It would be nice if we could spend more time on learning the C language in this booklet but we must press on. Try the books by PLUM, KOCHOW or others at your local Waldenbooks or B. Daltons bookstores. Each of these books is called 'Programming In C' or a similar title.

This implementation of C is a subset and does not contain some of the full C language elements:

The following operators are not supported:

. ? <-

The following preprocessor additions are available:

#define , #ifdef , #ifndef , #else , #endif , #asm , #endasm

#define defines a macro which must have a name but can be empty. For example:

```
#define mymacro a= b + c;
#define nullmacro
```

The commands #ifdef and #ifndef are used for conditional compilation.

```
#ifdef nullmacro
/* compile this code here if nullmacro was defined*/
/* code here */

#else
/* compile the following code */
/* code here */

#ifndef gloopy
/* compile if gloopy was NOT defined */
/* code here */

#endif /* must end with this command */
```

The #asm and #endasm commands allow you to add 6502 assembly language to your programs. This can be used for various things such as:

```
click {
   #asm
   LDA SPEAKER ; function to click the apple speaker
   #endasm
}
```

Neither labels or goto is supported directly. You will have

to use the #asm #endasm options and create your own assembly language jump if you absolutely have to use a goto. Remember that you may get unpredictable results if you don't fiddle around with the stack pointer. This entire compiler was written in Pascal and uses NO GOTOS. It can be done and is usually better and faster.

Missing elements include those dealing with the floating point math and the structure operations along with several keywords useful for describing variables when a machine has a more complex register set or instruction set. These are:

float , double , struct , union , long , short , unsigned
auto , register , typedef , static , goto , entry

One important caution is that the compiler uses a recursive descent equation parser which is very memory inefficient. What this means to the user is that for several layers of parentheses a stack overflow could occur. Use of the system swapping option from the command level should give you an extra 1k of memory space on the stack. Currently the limit is around 6-8 nested parens with the swapping option on.

The compiler will allow arrays of pointers and the calling of functions directly in memory such as:

```
1000(); /* calls a machine language subroutine at
           address 1000 decimal */
a=&func(); /* if a is a pointer */
a() /* calls function func */
```

The system assembler will not allow blocks of space larger than 399 decimal and will not assemble code with more than 5 or 6 of these in a row. The proper method to handle this is to define an array in a Pascal program and reference it using the public command. See the section on the linker for examples of this. Also look at the SIEVE programs on your compiler disk.

Under the APPLE Pascal operating system you are in the command mode after booting up. From the command mode you should type E for edit to create the C program file for the compiler. For information on the use of the editor please see **The APPLE Pascal Operating System Manual.**

The files you will need for the operation of the Thunder C Compiler are as follows:

```
SYSTEM.ASSEMBLER
SYSTEM.LINKER
SYSTEM.EDITOR
6502.ERRCODES
6502.OPCODES (* ALL OF THESE ARE FROM THE APPLE PASCAL
                SYSTEM AND ARE NOT SUPPLIED *)
C.CODE       (* THIS IS THE COMPILER SUPPLIED BY THUNDER
                SOFTWARE *)
```

Once you have typed in a program and are sure there are no mistakes (hopefully) you should save the file with a reasonable name that explains its purpose. Please don't name files A or TEST or QUMQUAT if they really are a word processor, a backgammon game and a super duper hi-res golf game. Now that you have a C program ... WAIT! do you have a program? NO? OK here is a program to use for your test.

```
int first,second,third;

adder() {
   first = 2;
   second = 3;
   third = first + second;
   printint(++third);
}
```

Allright, once you have typed in this program using the editor you are ready to compile it! Type X for execute and then C for the program called C.CODE (this is the compiler). The compiler asks you for the following information (in CAPITAL letters).

OUTPUT FILE:

Type the name of your output file. This file will contain the 6502 assembly language compilation of your program. The compiler

will add the .TEXT if you omit it.

MONITOR FUNCTION HEADERS<Y/N/RETURN>

A Y will print the first line of each function and #DEFINE to the system terminal. N and RETURN will not print these.

PAUSE ON ERRORS<Y/N/RETURN>

A Y will cause the compiler to pause after printing each ERROR message to the system terminal and wait for a keypress. N and RETURN will post the the message in the listing file and continue. You may hit the escape key to exit the compiler at this point.

BEEP AFTER EACH ERROR MESSAGE<Y/N/RETURN>

A Y will make that obnoxious beep after each message. N and RETURN will not make that obnoxious, odious beep.

LISTING FILE NAME: RETURN FOR NONE

You may type in a file name here to contain the listing of the program. This may be the same as the output file typed in above. If so it must be typed with the .TEXT tacked on the end of the name.

In all cases where you are asked for a file name you may use STDIN for the keyboard as input, STDOUT as the system terminal and any legal Pascal identifiers such as PRINTER:, SYSTERM:, #4:TESTFILE.TEXT and so on.

The next question you must answer is

ENTER NEXT INPUT FILE NAME:

Puzzled? You haven't entered the first input file name! Well, you are now in the compilation loop and thus may enter a second, third etc. file name to be compiled one after each other. If you have not got another file to input you may type RETURN and end the compilation session. On the first pass, though, a RETURN will default to STDIN to allow you to enter programs from the keyboard. Be careful though because there are no editing facilities. Once again the compiler will add the .TEXT if you omit it.

Lastly the compiler must ask for housekeeping information for the assembler.

NAME OF THIS MODULE?<DEFAULT MAIN>

A module is a seperately compiled group of C functions. This is the method for creating libraries of previously compiled functions. You will have to define an external reference if a function is to be called across a module boundary. In general you should name the module sensibly but you may not use a name that will be used later as a function name. The Pascal program will reference your program by this name.

INCLUDE A LIBRARY? <NAME/N/DEFAULT STDIO.H, UTILITY.H>

The library headers only need to be included in the modules that call functions in that library. Answer N to stop adding libraries.

Now the compilation will commence. During the compilation you will see a message printed for each line of source (perhaps more) that consists of the line number, the nesting level and the amount of available memory at that node point.

If you used the above example you should now have a file named output.text (or whatever you called it). You now need to assemble your program. First, though, you should read through the sections entitled THE STDIO LIBRARY and THE UTILITY LIBRARY.

C has associated with it a library of functions that perform various input and output operations. Unlike other languages which actually include i/o as part of the language definition, as in FORTRAN or BASIC, C uses functions that interact with the host machine. Since we are not trying to imitate UNIX on our trusty little APPLE // we may not be exactly the same. Some of the routines that are or will be supported by the THUNDER C COMPILER are as follows:

```
open(filename,id) - open the file and assign a descriptor.
                    returns id. Filename is a string.
close(id) - close the file associated with id.
           returns nothing.
fgetc(c,id) - get the next character from the file id.
             returns c.
fputc(c,id) - put the next character on the file id.
             returns nothing.
fgets(string,length,id) - get a string from a file id.
                          returns length.
fputs(string,length,id) - put a string on the file id.
                          returns nothing.
getchar(c) - get a character from the keyboard.
            returns character.
putchar(c) - put a character on the system terminal.
            returns nothing.
```

To use any of these functions in your programs you may include the system file STDIO.H which contains the linker information needed to link the file STDIO.CODE to your final program. You may instead include the source code for any of the routines in with your program file to elimanate the functions that you are not using. These routines will allow you to interact with the system. Other routines will be included in future releases of the compiler system. Examine the library to see what is available in your version. These routines may be subject to change, modification and non-availability depending on the release level of the compiler. If you wish to write any routines for a library we will be happy to review them for inclusion here.

Examples of use:

```
/* open a file */
status = open("system.wrk.text",&fd);y
/* close that file */
status = close(fd);
/* get a string from system.wrk.text */
len = fgets(&strptr,maxlength,fd);
```

The utility library is where you may store your utility routines and where various functions may be available. These may include the following:

gotoxy(x,y) - moves the cursor to position x vertically and y horizontally.

printint(i) - prints the integer i on the screen.

hexprint(i) - prints the integer i as a hexadecimal number on the screen.

getblock(0,unit,buffer,length,blocknum) - gets the block from a Unit and places it in the area pointed to by buffer. The length is in bytes and the blocknum is the number on the unit. For example if one wanted to read the directory block number 2 s/he would call getblock(0,0,&array,512,2).

putblock(0,unit,buffer,length,blocknum) places the material stored in the area pointed to by buffer onto the unit at the specified block number blocknum. For example, if you had just read the block as shown in the example under getblock you would call the following to place it back on the disk.

```
putblock(0,0,&array,512,2)
```

The zero is a dummy value.

To use any of these routines you must include the file UTILITY.H which contains the header information used to link the programs together. The compiled source file is named UTILITY.CODE. If you happen to write any routines you would like to share, please send them in for review and possible inclusion in this library.

To add functions to either of these libraries you must put the source code into the proper source file and recompile it with the proper .DEF option. Then you must change the proper .H file so that you may reference it using the header method.

Now that you have read the last two sections you are ready to assemble your program. First, type A for assemble from the command level and answer the question about the input file with your filename. Then, for the output of the assembler you should type the name using the size definition as follows:

```
output.code[15]
```

or use the automatic name

```
system.wrk.code
```

by typing <RETURN>. The space definition is necessary when not using system.wrk.code because of the fashion of file handling that the assembler uses (see the addendum to the operating system manual). While the assembler is assembling your file you shouldn't get any messages of problems. Unless, of course, something is missing like the RUNTIME.H.TEXT header file, which includes the definitions for the runtime library code file, or the files STDIO.H.TEXT and UTITLITY.H.TEXT which contain definitions for the library code files you may have called.

Upon completion of the assembly you should be told that there were no errors. It will now be time to link your file to a Pascal program that calls an external procedure MAIN. See the section on the linker to find out how to do this.

You may edit your compiled file before assembly to add features or do some hand optimizing. You may also change some things such as adding macros or other such nifty items. This is completely at your own risk and we are not responsible for your experimentation. If you are interested in writing an optimiser for crunching the code files we would be interested in talking to you.

IMPORTANT NOTES

THE ASSEMBLER WILL NOT ALLOW YOU TO USE THE SYMBOLS A , X OR Y AS GLOBAL VARIABLES.

The APPLE Pascal linker is a much misunderstood portion of the system. This program connects all of the loose ends that separate compilation of programs causes and prepares the file for execution. It is not difficult to use and can even be controlled by an EXEC file which is how we do it. An example of an EXEC file is on the disk and is named LINK.TEXT. To use this you should type EXEC/LINK from the command level and have the files

RUNTIME.CODE,
STDIO.CODE,
UTILITY.CODE,
DUMMY.CODE and
SYSTEM.WRK.CODE (your C program already assembled)

on line on the boot disk. All of the linking operation will proceed without further input from you.

It is also possible to do the linking yourself if you have any special files or different library names. A dummy Pascal program is supplied for you to use as an object to link to. The file name is DUMMY.CODE. To use this program invoke the linker using the L option from the command level and type in the filename DUMMY in answer to the first prompt.

Next type the name of your assembled code file generated by the compiler.

Follow that by RUNTIME in answer to the next prompt.

Next type STDIO and then UTILITY if you are using any of the functions from either of the two libraries.

The linker will read all of the files and then ask for an output file name. This will be the executable code file of your program typed in way back when. Once this is done you may execute the file using the X for execute command from the command level and then typing the name of the executable code filename.

If your program won't link you can try several things. First of all, the linker can get more space by creating a file SYSTEM.-SWAPDISK[4] and by invoking the system swapping option. Second, make sure that all of the Pascal procedures you have in your program are nested instead of global. This takes less space in the symbol tables of the procedures. We had some difficulty for this compiler with around 3500 lines of code. You shouldn't have difficulty unless you are making programs that long. You can also try incorporating multiple files for linking into a library file and link the one library at link time.

The APPLE Pascal system allows you to write 6502 programs and link them to P code programs using the linker facilities. There are a number of considerations you must examine when doing this. Some of these are that you may access global variables from the Pascal program and that an external program may be called as a function or a procedure. The compiler allows several options to cover some of the linking capabilities. The external statement has been expanded to include definition of external variables or functions, reference to external variables or functions and the access to global or public information from the Pascal program. These will be necessary to properly link to your own Pascal programs. The syntax is as follows:

```
external ref a[100]
external def glop,func1
external public array1[1200]
```

Note that all external references are integer and may not be of type character.

You will also need to use the public definition if you need more than 1000 words of memory for an array. Study the SIEVE programs on your disk to see how this was done. The compiler supports this under the external command as follows:

```
external public a[5000];
/* defines .PUBLIC A in code stream */

external a;
/* defines .REF A in code stream */

external def a();
/* defines .DEF A in code stream */
```

This extension is NOT STANDARD C and is for use with the APPLE Pascal assembler and linker only. The public option allows you to access values from a Pascal Main programs global variable area. The REF and DEF options allow you to reference and define global variables between seperately compiled C programs. For example, to reference an external function funcext you would define funcext in the source file that contains the funcext program using the external def command and refer to it in another file by using the external ref command as follows.

```
external def funcext();

funcext() { your function }
/* end of the first file */
-----------------------------------------

external ref funcext();

otherfunc() { int var;
var = funcext();
}   /* end of the second file */
```

The connecting of the missing information will be handled by the linker when you link all of the seperately compiled files together.

These functions that you write in C can be used as high speed functions for Pascal; For example Pascal took over 12 seconds to print the alphabet 100 times and C took slightly more than 2 seconds! It is planned that we will use C subroutines to help speed up our compiler in this fashion.

In your Pascal programs you will need to declare the C program as an external procedure as follows:

```
program pascal;

var bigarray : array [1..1024] of integer;

procedure main;
external;

(* your other procedures *)

begin (* pascal main block *)
      (* other calls *)
      main; (* calls your c program *)
end.
```

You will need to have a program compiled by the C compiler with the name main as the first function in the file such as the following:

```
external ref bigarray[1024];
main() { your c program }

/* other c programs */
```

To execute your compiled and linked program you merely type X from the command level and then the code file name of the linked program.

This compiler supports the following elements of the C language:

if , else , while , do , break , char , int , extern , continue
for , case , default , switch , return

assignment operations = += -= ^= &= *= %= <<= >>= != /=

unary operations * & - ! ~ ++ --

binary operations * / % + - >> << < > <= >= == != & ^ | && ||

and the comma , operator.

Pointers are supported and address calculation is supported.
The goto statement is not supported nor are statement labels.
Identifiers are unique to 8 characters but may be of any length.

A list of statement formats are as follows:

```
compound statement
expression;
if (expression) statement
if (expression ) statement else statement
while (expression) statement
do statement while (expression);
for (expression1;expression2;expression3) statement
switch (expression) statement
case constant-expression : statement
default : statement
break;
continue;
return;
return expression;
;
```

The APPLE][series computers, until the release of the APPLE //e, have had some missing keys on the keyboard. Because C makes heavy use of these missing keys, a method had to be developed to make up for them. This and the standard C preprocessor facilities are available in our preprocessor. The following preprocessor macros are available to add the APPLEs missing keys:

Missing key	name	what to type
\	backslash	//
~	tilde	/tilde
!	bar	/bar
{ }	brackets	type (* and *) as in Pascal or use /lbracket and /rbracket.

The following are available for conditional compilation and macro handling:

#ifdef macroname	compiles following lines if defined
#define macroname	defines a macro
#ifndef macroname	compiles the following lines if not defined
#else	compiles if previous control statement was false
#endif	ends a block of conditional compilation

In string routines it is necessary sometimes to use the explicit representation of unprintable characters such as form feed, newline etc. These are usually done using the backslash. In this compiler you may need to type // instead of \ which is not normally available. Those who have an APPLE //e may use the normal backslash character. The following characters are available in strings (enclosed in double quotes i.e. "abcd"):

newline	//n
formfeed	//f
tab	//t
backspace	//b
octal	//*** where *** is three octal digits

All keywords must be in uppercase, unlike most of the examples in this book. Lowercase letters may be used for variable names only but it is not recommended because it just makes the editing of programs more difficult. Future versions of the compiler may handle upper and lower case keywords.

```
#DEFINE ISALPHA(C) (C>='A') && (C <= 'Z')
/* A USEFUL MACRO */
MAIN() (* CHAR C;
   C = 'A';
   WHILE (ISALPHA(C)) PUTC(C++);
   /* SHOWS THE USE OF A MACRO ISALPHA(C) AND A STDIO FUNCTION
      PUTC TO PRINT THE ALPHABET */

   FOR (C='A';C<='Z';C++) PUTC(C);
   /*  SHOWS  THE  USE OF THE FOR STATEMENT TO PRINT THE ALPHABET
   */

   C = 'A';
   DO PUTC(C++) WHILE (C<='Z');
   /* USE OF THE DO WHILE STRUCTURE TO PRINT THE ALPHABET */

   SWITCH ( C ) (*
      'A': PUTC('A'); BREAK;
      'Z': PUTC('Z'); BREAK;
      DEFAULT: PUTC(' '); BREAK;
      *)
   /* PRINTS THE LETTER IF IT IS AN  'A'  OR  A  'Z'  AND  PRINTS
      BLANK IF IT IS ANYTHING ELSE */

   IF (C == 'Y') PUTS("YES\N");
   ELSE PUTS("COMO?\N");
   /*  PRINTS  THE  STRING YES IF THE CHARACTER IS A 'Y' AND ASKS
      THE QUESTION COMO? IF IT IS ANYTHING ELSE */

   (* INT A , B , C ; /* DEFINES BLOCK LOCALS A,B,C */

      A += 1;
      A = A + 1;
      A++;
      ++A; /* faster!! */
      /* FOUR WAYS OF STATING THE SAME EQUATION */

      A = INTFUNC(B,C); /* ASSIGNS THE VALUE OF  INTFUNC  TO  A*/


      A = B & C; /* BITWISE AND OF B AND C */

      A = B && C; /* LOGICAL (TRUE OR FALSE) AND OF B AND C*/

      *) /* CLOSES OUT THE BLOCK */
   *) /* CLOSE OUT THE FUNCTION */
```

A compiler is merely a series of programs that translate a program into another form. Some compilers translate a language such as RATFOR into another language such as FORTRAN. In this case we may call the RATFOR translator a PREPROCESSOR. In reality, though, it is a compiler. There are preprocessor type compilers available for all sorts of purposes such as RATFOR or a structured BASIC for APPLESOFT. Examine the following model of a program line.

```
[LABEL] STATEMENT
        STATEMENT = KEYWORD
                  = FUNCTION CALL
                  = EXPRESSION
                  = KEYWORD (EXPRESSION) STATEMENT
```

This is not a complete specification of a programming grammar but it will suffice as a simple example for your information. The statement label is optional and is not used in all languages. On the other hand, BASIC uses a label at all times. If you were a compiler you would need to know the above information to be able to translate the language properly. So the first thing is to specify the language completely and then to begin translation. Here follows a simplified pseudo program similar in operation to the C compiler.

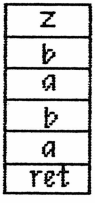
```
REPEAT BEGIN
   DOGLOBALS (* DEFINE GLOBAL VARIABLES *)
   NEWFUNCTION (* NEW FUNCTION DEFINITION *)
      BEGIN
         SYMBOLNAME  (*  CHECK  FOR  ALPHA  NUMERIC  BRACKETED BY
                        BLANKS OR SPECIAL CHARACTERS *)
         MATCH('(') (* NEED THIS PAREN *)
         DOARGS (* ANY ARGUMENTS? *)
         MATCH(')') (* CHECK FOR END PAREN *)
         ARGDEFINITIONS (* ONLY IF THERE WERE ARGS *)
         STATEMENT
            MATCH('{') (* NEED THIS BRACKET *)
            DOLOCALS (* DEFINE LOCAL VARIABLES IF ANY *)
            REPEAT BEGIN
               KEYWORD (* THINGS LIKE IF, WHILE, DO, BREAK ... *)
               EXPRESSION (* SCANS FOR SYMBOLS AND ASSIGNS REGISTERS
                           FOR ALL OPERATIONS *)
                  REPEAT BEGIN (* RECURSIVE DESCENT PARSER *)
                     HEIR1    (*  each   of   these   handles
                      :                a group of operations
```

```
                    HEIR14    of the same precedence*)
                    PRIMARY (* scans for a variable name *)
                  END (* EXPRESSION *)
                  UNTIL MATCH(';')
            UNTIL MATCH(')')
      END (* NEWFUNCTION *)
END (* MAIN LOOP *) UNTIL ENDOFDATA
```

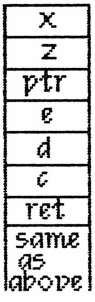
Memory map of C function calls

When a C function is called by another C function the arguments, if any, are placed on the stack in left to right order. When in a C function a new block is begun the programmer may add new local variables which are placed on the stack in left to right order. If, for example, we had a series of functions as follows, the diagrams on the right will show the condition of the stack.

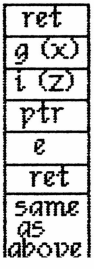
```
INT alpha,beta;
CHAR zeta,zero[100];
main() { INT a,b;
      :
      :
   func(a,b,z);
      :
      :
```




```
func(x,y,z) int x,y; char z; { INT c,d,e;
      :
      :
   otherfunc("hello there",z,x);
      :
      :
   }
```




```
otherfunc(p,i,g) char *p; INT i,g; {
      :
      :
      :
   }
```



Thunder Software
P.O. Box 31501
Houston, Texas 77231
(713) 728-5501